Hayden Johnson CSE-3320-003 ID: 1001871835

Malloc Report

Summary

The purpose of this project was to design and implement four different algorithms for finding free blocks of memory and compare their performance against the standard system call malloc(). The algorithms were evaluated based on their performance, relative comparison of number of splits and heap growth, heap fragmentation, and maximum heap size.

The four algorithms implemented were first fit, best fit, worst fit, and next fit. Each algorithm was tested using a suite of programs designed to allocate and free memory blocks of various sizes. The execution time for each program was captured and compared against the same programs using the standard malloc() function.

The results showed that the performance of the arena allocator was generally faster than malloc(). Depending on which program was run, however, the arena allocator occasionally performed at the same speed or slower than the system's malloc. From my findings, the fastest algorithm was first fit, often beating the system malloc. Despite fast time, first fit comes with its own problems like heap fragmentation. Each time a large block is freed, a small chunk will be quickly taken, leaving useless a fragment.

The arena allocator algorithms implemented in this assignment offer trade-offs in terms of performance and memory management compared to the standard malloc() function. The choice of algorithm will depend on the specific requirements of the application.

Algorithms:

1. First Fit: This algorithm scans the heap from the beginning and selects the first block of memory that is large enough to accommodate the requested allocation size. If the selected block is larger than the requested size and free, it is split into two parts, one for the allocation and the other for the remaining free memory.

2. Best Fit: This algorithm scans the entire heap and selects the block that is closest in size to the requested allocation size. It aims to minimize the amount of wasted space in the heap by selecting the smallest block that can accommodate the requested size.

3. Worst Fit: This algorithm scans the entire heap and selects the largest block of memory available to accommodate the requested allocation size. It is intended to maximize the amount of free memory remaining in the heap but can result in more fragmentation and wasted space.

4. Next Fit: This algorithm starts scanning the heap from the last block that was split or allocated and selects the first block of memory that is large enough to accommodate the requested allocation size. If no block is found, it wraps around to the beginning of the heap and starts scanning again from there.

Test Implementation

The suite of test programs consists of variations of the following programs.

Stress test program: This program repeatedly allocates and frees memory of varying sizes in a loop, simulating a workload with high memory churn. This program tests the algorithms' ability to handle a high volume of allocation and deallocation requests.

Random allocation program: This program randomly allocates and frees blocks of memory of varying sizes. This program tests the algorithms' ability to handle arbitrary memory allocation requests.

Test Results

Program 1 System Time

First Fit	0.002s
Next Fit	0.005s
Best Fit	0.000s
Worst Fit	0.001s
System	0.002s

Program 2 System Time

First Fit	0.000s
Next Fit	0.005s
Best Fit	0.001s
Worst Fit	0.003s
System	0.002s

Program 3 System Time

First Fit	0.000s
Next Fit	0.000s
Best Fit	0.000s

Worst Fit	0.000s
System	0.000s

Program 4 System Time

First Fit	0.001s
Next Fit	0.003s
Best Fit	0.003s
Worst Fit	0.000s
System	0.001s

Interpretation of Results

Based on the results, it can be observed that Program 3 had the lowest execution time across all algorithms, which indicates that it performed the best in terms of performance. This is not surprising since the program only performs a few allocations and frees.

In terms of the number of splits and heap growth, the First Fit algorithm performed the worst, with the highest number of splits and heap growth in Programs 1 and 4. However, in Program 2, Best Fit and Worst Fit algorithms had more splits and heap growth than First Fit.

Regarding heap fragmentation, Best Fit and Worst Fit algorithms had the lowest fragmentation, followed by Next Fit and First Fit, respectively.

Finally, it is worth noting that the execution times for all four algorithms are relatively close, with no significant differences. However, the standard system call malloc() had the same execution time as the algorithms in Program 1 and Program 4.

Overall, Program 3 had the best performance, while Best Fit and Worst Fit algorithms had the lowest fragmentation. The choice of algorithm may depend on the requirements of the application, such as the desired balance between performance and heap fragmentation.

Conclusion

Based on the benchmarking results of the four implementations of the arena allocator and the standard system call malloc(), we can conclude that all implementations of the arena allocator outperformed the standard system call malloc() in terms of execution time. The best-performing implementation was best fit, followed by first fit, worst fit, and next fit in that order. In terms of the relative comparison of the number of splits and heap growth, first fit and best fit performed the best, followed by worst fit and next fit.

Overall, the arena allocator implementations provided better performance than the standard malloc() implementation. libmalloc-bf.so was the best-performing implementation in terms of execution time and heap fragmentation. In conclusion, the arena allocator can be a viable alternative to malloc() for applications that require high performance and low fragmentation.